Result Invalidation for Incremental Modular Analyses

Jens Van der Plas[0000−0002−7475−576X], Quentin Stiévenart[0000−0001−9985−9808], and Coen De Roover[0000−0002−1710−1268]

Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
{jens.van.der.plas,quentin.stievenart,coen.de.roover}@vub.be

Abstract. To reduce the running time of static analysis tools upon program changes, incremental static analyses reuse and update pre-existing results. Such analyses must efficiently detect and remove outdated results. We introduce three novel, complementary result invalidation strategies for incremental modular analyses. The core idea of our work is to alternate invalidation with computation. We apply our strategies to a recent, state-of-the-art incremental modular analysis that suffers from imprecision, and evaluate them on soundness, precision, and performance. Our strategies lead to precision improvements compared to an incremental analysis without invalidation, though the precision of a full reanalysis is not yet matched. On most benchmarks, our incremental analysis performs well. However, on some benchmarks our analysis performs poorly as the changes drastically change program behaviour, for which the changes are difficult for an incremental analysis to handle.

Keywords: Static program analysis · Incremental program analysis · Modular program analysis.

1 Introduction

Static analysis is an approach to computing properties of programs without running them. It is the foundation of code smell, bug, and vulnerability detection tools (e.g., [15, 35, 21, 14, 28]) used in modern software engineering processes such as continuous integration pipelines [31]. An analysis that is fast in the presence of small and frequent code changes can even be incorporated into a development environment. To meet these demands, incremental static analyses have been proposed [2, 27, 24, 8, 13]. Given the results of an *initial analysis*, an *incremental analysis* updates the results given the code changes. The goal of an incremental analysis is to produce results faster than a *full reanalysis* by reusing and updating previous results.

Recently, Van der Plas et al. [18] introduced a general approach to rendering any modular static analysis incremental. Modular analyses divide a program into parts which are (re-)analysed separately but whose analyses may be interdependent. The authors posit that modularity facilitates bounding the impact of changes. While the evaluation shows that incremental updates are often faster

than a full reanalysis, incremental updates may be less precise than a full reanalysis as the presented analysis cannot delete outdated results. In this paper, we improve upon the approach by Van der Plas et al. [18] as follows:

– We introduce three complementary strategies to regain lost precision. The idea is to *interleave* invalidation with recomputation, to maximise reuse of previously computed results. Our strategies can be applied to modular static analyses that employ global-store widening and infer dependencies amongst components.
– We implemented these strategies and evaluate their impact on the precision and performance of the incremental analysis, when used alone or in combination.

2 Background

We now introduce modular static analysis, following a recent formulation by Nicolay et al. [16]. We obtain an incremental version of this formulation by applying the incrementalisation approach by Van der Plas et al. [18].

2.1 Modular Static Analysis

A modular static analysis [5] divides a program into static parts, e.g., function definitions, referred to as *modules*. A module may have multiple runtime instantiations, e.g., function calls, which the analysis might discern as well. We refer to their reification in the analysis as *components*. A component consists of a module and a context used to discern the different instantiations. Depending on the definition of contexts, more instantiations may be discerned, increasing the analysis precision (and complexity). A modular analysis analyses its components in isolation. The analysis of one component may however trigger the (re-)analysis of another. The remainder of this paper focuses on function-modular analyses. All examples use a lattice representing each value as a set of its possible types, and empty contexts, i.e., every module will correspond to at most one component.

Effect-Driven Modular Static Analysis Recently, ModF, an effect-driven formulation of function-modular analysis has been introduced [16]. ModF is a control-flow analysis also computing value information. It *reifies* the *computational dependencies* between components and uses these to drive a fixed-point computation, alternating between an *inter-component analysis*, scheduling components for analysis, and an *intra-component analysis*, analysing individual components. The inter-component analysis, referred to as Inter and shown in Alg. 1, uses a worklist of components to be analysed. Initially, this worklist contains a Main component[1], representing the program's entry point (line 1). Every analysis step removes a component from the worklist (lines 6-7) and analyses it (line 8); the analysis terminates when the worklist is empty.

[1] In formalisms, lowercase Greek letters denote components (e.g., α and β). Otherwise, we denote them by their corresponding name in small caps (e.g., Main and fib).

Algorithm 1: The inter-component analysis (INTER) of ModF.

```
1  WL := {Main}; // The work list, initially containing the MAIN component.
2  V := ∅; // The visited set.
3  D := λr.∅; // Set of dependencies (read effects).
4  σ := λa.⊥; // Global value store, initially all addresses map to bottom.
5  while WL ≠ ∅ do
6  |  α ∈ WL;
7  |  WL := WL \ {α};
8  |  (C′, R′, W′, σ′) = intra(α, σ); // Intra-component analysis.
9  |  σ := σ′;
10 |  V := V ∪ {α};
11 |  WL := WL ∪ (C′ \ V);
12 |  foreach r ∈ R′ do D := D[r ↦ D(r) ∪ {α}];
13 |  foreach w ∈ W′ do WL := WL ∪ D(w);
14 end
15 return (σ, V, D);
```

The *store*, mapping abstract addresses to abstract values, abstractly represents the heap. ModF uses global-store widening [30], i.e., there is a single global value store σ within the analysis [16]. For every component, σ contains an abstract return value. Upon a function call, ModF does not step into the function, but retrieves the stored return value (or $\bot$ if no value had been stored).

A component's analysis returns a set of *effects* reifying its computational dependencies, together with an updated store (line 8). Dependencies are function calls (generating *call effects*) and reads/writes in the store (generating resp. *read/write effects* – the latter is only generated when σ actually changes).[2] These effects are used to determine the component(s) to be added to the worklist, causing components depending on updated information to be reanalysed.

A ModF analysis results in (line 15): (1) the store σ, (2) the set of components created, and (3) the set of dependencies (read effects). We consider all parts of the result equally relevant, though in practice one might only be interested in σ.

Example. We illustrate how ModF computes the control-flow and value information of the Scheme[3] program in Listing 1. ModF analyses it as follows (omitting some effects for brevity):

1. The analysis starts with MAIN. Binding `x` generates a write effect for this variable. Then, a call effect to `fun` is generated, and the corresponding component, FUN, is added to the worklist. As no return value had been computed for FUN, $\bot$ is read from the store; a read effect on this return value is registered.

2. FUN is analysed, producing a call effect for `inc` and read effects for `x` and for the return value of INC. The new component INC is added to the worklist, as is MAIN because FUN's return value is updated to `Int`, generating a write effect.

[2] For brevity, in pseudocode, the set C represents the set of all components corresponding to the emitted call effects, and the sets R and W represent the addresses corresponding to the emitted read and write effects respectively.

[3] In this work, we use Scheme, a dynamically-typed dialect of Lisp with support for higher-order functions. Its dynamic nature makes it difficult to analyse as control and data flow are intertwined, precluding the computation of a call graph ahead of time. Scheme is representative for a whole class of languages such as JavaScript.

```
1  (define x 0)                          ; Definition of a variable x.
2  (define (fun) (inc) x)                ; Function that reads x.
3  (define (inc) (set! x (+ x 1)) #t)    ; Function that reads and writes x.
4  (fun)
```

Listing 1: Example Scheme program of two functions.

3. Either MAIN or INC can now be analysed. Assuming INC is analysed (the order does not affect the result [16]), INC reads `x`, generating a read effect, and also writes to this variable. As the value in the store is not updated, no write effect is generated. As the return value of INC is updated to `Bool`, a write effect is generated and FUN is added to the worklist again.
4. The analysis continues until the worklist is empty.

The principle of effect-driven flow analysis is applicable to different module granularities, e.g., thread-modular analyses [22], and can be used with any abstract domain without infinite ascending chains and with any context-sensitivity.

The Component Graph The analysis of a component generates call effects, each corresponding to a component discovered by the analysis. After the analysis of a component α, INTER collects the set of components *called* by α, denoted C_α. This gives rise to a cyclic directed graph, the *component graph*, representing how components are created: for every component $\beta \in C_\alpha$ there is an edge from α to β. Fig. 1 depicts the component graph from previous example.

Fig. 1: The component graph corresponding to the analysis of the program in Listing 1: `inc` is called from `fun`, which is called from the program's entry point.

2.2 Incremental Modular Static Analysis

Van der Plas et al. [18] present an approach to rendering an effect-driven modular static analysis incremental. It requires the analysed program to be annotated with *change expressions*, which are akin to the patch annotations of Palikareva et al. [17]. A change expression specifies how a given expression is updated. Its first argument represents the original expression; its second argument represents the expression that replaces the original. Change expressions can be added manually, or be inserted by a change distiller (e.g., [7, 6]) or change logger (e.g., [32, 10, 12]). In the following function, the predicate is updated from `(= n 0)` to `(< n 2)`:

```
1  (define (factorial n)
2    (if (<change> (= n 0) (< n 2))
3        n
4        (* n (factorial (- n 1)))))
```

For a given set of change expressions, Van der Plas et al. [18] compute the affected analysis results and update them accordingly. Their analysis tracks which change expressions within the source code of a module were encountered during the analysis of the corresponding components. Every component whose analysis encountered a change expression is considered to be *directly affected*. If an expression in a module changes, only the components that encountered this expression during their analysis are affected. All directly affected components are added to the worklist and the fixed-point computation is restarted. The modular analysis design ensures that indirectly affected components are reanalysed too.

Sources of Imprecision Tab. 1 shows the three parts of the result of a ModF analysis. The approach by Van der Plas et al. [18] only updates prior results monotonically: no outdated information can be removed; the result of the analysis over-approximates the behaviour of both the updated and original program. All parts of the result may suffer from imprecision, as shown in Tab. 1. This means that components and dependencies no longer representing the program's behaviour cannot be removed. In σ, values cannot become more precise. Imprecision in one part of the result may cause imprecision in other parts. E.g., when a value in σ is imprecise, the analysis may explore more paths and thus infer more components and dependencies, which may in turn degrade the store's precision.

3 Strategies for Precision Recovery

We now introduce three complementary strategies that improve the precision of an incremental analysis result by invalidating the information that corresponds to outdated program behaviour. The aim is to minimise the precision loss caused by monotonic updates to a prior analysis result, without increasing analysis time.

3.1 Invalidation Principle

The presented strategies treat the intra-component analysis as a black box and do not put any restrictions on the lattice nor on the context-sensitivity used by the analysis. The intra-component analysis must only compute a set of effects.

The aim is to invalidate as few valid results as possible, so that results not impacted by a change need not be needlessly recomputed. Related work [2, 13] often consist of an *invalidation phase*, which *over-approximates* and clears outdated results, and a *recomputation phase*, which updates the analysis results. To avoid over-approximating outdated results, we *interleave* invalidation with recomputation, maximising reuse. After an intra-component analysis, Inter computes which parts of the results have become obsolete and removes them; information is only removed when it is no longer computed by an intra-component analysis. Mapping this onto Alg. 1, invalidation happens after line 8. Our approach leads to a recompute-and-invalidate cycle: the analysis of a component may lead to a result invalidation, which in turn can lead to more analyses of components.

Table 1: Overview of the parts of the analysis result, of the sources of imprecision for each part, and of the corresponding strategies to invalidate outdated results.

Components	
Explanation	Set of components created during the analysis, each abstractly representing an aspect of the runtime behaviour of the program, e.g., a function call.
Imprecision	Components no longer representing the program's behaviour cannot be removed.
Solution	Component Invalidation (CI): remove components that are no longer created.
Dependencies	
Explanation	Set of inter-component dependencies (read effects) computed during the analysis, each marking a link between a component and an address in the global value store σ. Using these dependencies, the analysis of one component takes into account information computed by the analysis of other components.
Imprecision	Dependencies that are no longer valid cannot be removed.
Solutions	Dependency Invalidation (DI): remove dependencies that are no longer computed by the reanalysis of an impacted component. CI: removing a component clears its dependencies.
Value Store σ	
Explanation	Over-approximates the heap. Mapping of abstract addresses to abstract values.
Imprecision	Values in σ are updated monotonically, since they are joined upon updates.
Solutions	Write Invalidation (WI): improve the precision of values in the store σ by removing values that are no longer written. CI: when WI is enabled, the removal of a component may allow σ to be refined.

```
1  (define (fac n)
2    (if (< n 2)
3        n
4        (* n (fac (- n 1)))))
5  (define (fac-loop n) ; Executes the ‘fac‘ function in a loop.
6    (define (loop i)
7      (if (< i n)
8          (begin
9            (display (fac i))
10           (display " ")
11           (loop (+ i 1)))))
12   (loop 0))
13 (<change> (fac-loop 10) (fac 10)) ; Updated to call ‘fac‘ directly.
```

Listing 2: A change causing components to be removed.

Tab. 1 outlines the developed strategies, one for each part of the analysis result: component invalidation, dependency invalidation, and write invalidation. Though, invalidations in one part of the result may impact the other parts.

3.2 Component Invalidation (CI)

Component invalidation (CI) removes components from the analysis result that are no longer created by any other component, plus the dependencies related to these components. Consider e.g., the program in Listing 2. The initial analysis creates four components, shown by the component graph on top of Fig. 2. The change expression replaces the call to `fac-loop` by a call to `fac`; `fac-loop` (and transitively `loop`) are no longer called. The reanalysis of MAIN now finds that FAC-LOOP is no longer called: FAC-LOOP and LOOP can both be removed.

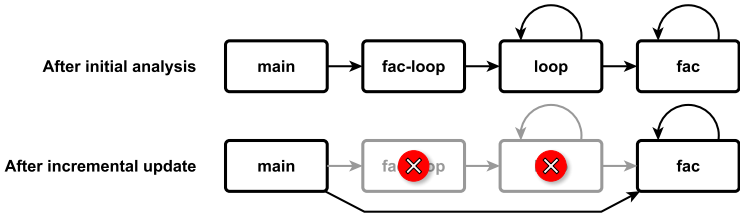

Fig. 2: ModF components for the program in Listing 2. On top, the components after the initial analysis of the program; at the bottom, the components after the incremental update. Arrows depict generated call effects.

CI uses the component graph to detect outdated components: all components no longer transitively reachable from MAIN, i.e., the entry point of the program, can be removed. Alg. 2 extends INTER with CI. For every component α, INTER caches C_α, the set of components called by α's last analysis, using a cache $\mathbb{C}$. The set of dependencies R_α, cached in $\mathbb{R}$, allows the efficient removal of the

dependencies of deleted components ($\mathbb{R}$ holds the same information as D but in the reverse order, avoiding a full traversal of D). After the analysis of a component α, the set of components called by the analysis of this component, C'_α, is returned. INTER then retrieves C_α, the set of components called during the *previous* analysis of α, and updates the cache $\mathbb{C}$ (lines 12-13). It then computes the set containing all components that are no longer called by α. If this set is non-empty, one or more edges were removed from the component graph and some components may have become outdated (line 14). In this case, the transitive closure of $\mathbb{C}$ is computed, starting from MAIN; all components that are not part of it are removed (lines 15-16). All dependencies of these components are removed too, avoiding the existence of dependencies to non-existent components. The transitive closure is needed because a component can only be removed if it is no longer created by any other component. Finally, $\mathbb{R}$ is updated (line 18). Note that lines 15 and 16 will never be executed during the initial analysis of the program. To avoid the needless but possibly expensive computation of set differences in the condition, we first check whether an incremental update is taking place (line 14). For similar reasons, we do the same for DI and WI.

Algorithm 2: INTER extended with component invalidation (in blue) and dependency invalidation (in purple).

```
   // Assumes the existence of a cache for the sets C, ℂ, initialised as ℂ := λα.∅
      before the initial analysis, and the existence of a cache for the sets R, ℝ,
      initialised as ℝ := λα.∅ before the initial analysis.
 1 Function deleteComponent(β) is
 2 |   foreach r ∈ ℝ(β) do D := D[r ↦ D(r) \ {β}]; // Delete dependencies.
   |   // Remove β from all data structures.
 3 |   V := V \ {β}; WL := WL \ {β}; ℝ := ℝ \ {β}; ℂ := ℂ \ {β};
 4 end
 5 while WL ≠ ∅ do
 6 |   ... // Ditto Alg. 1.
 7 |   foreach w ∈ W' do WL := WL ∪ D(w);
 8 |   if incremental update then
 9 |   |   R := ℝ(α);
10 |   |   foreach r ∈ (R \ R') do  D := D[r ↦ D(r) \ {α}];
11 |   end
12 |   C := ℂ(α);
13 |   ℂ := ℂ[α ↦ C']; // Update ℂ immediately to use the updated C'.
14 |   if incremental update and C \ C' ≠ ∅ then
15 |   |   reachable := ℂ(Main) ∪ {β | γ ∈ reachable ∧ β ∈ ℂ(γ)};
16 |   |   foreach β ∈ (V \ reachable) do  deleteComponent(β);
17 |   end
18 |   ℝ := ℝ[α ↦ R']; // Both for component invalidation and dependency invalidation.
19 end
20 return (σ, V, D);
```

3.3 Dependency Invalidation (DI)

The second strategy, *dependency invalidation* (DI), removes outdated dependencies. This ensures that components are not spuriously reanalysed. Consider, e.g.,

```
1 (define x 1)
2 (define y 2)
3 (define (write) (<change> (set! x 7) (set! y 7)))
4 (define (read) (<change> x y))
5 (read)
6 (write)
```

Listing 3: Example program with changing dependencies. Initially READ has a dependency on the address of variable `x`, a_x. In the new version of the program, READ solely has a dependency on a_y, the address of variable `y`.

```
1 (define (fromBool b)
2   (if b
3       (<change> 'aSymbol       "aString")
4       (<change> 'anotherSymbol "anotherString")))
5 (define x (fromBool (some-complicated-predicate)))
6 (display x)
```

Listing 4: Example program. Initially, `x` only holds a symbol, whereas after the update it can only contain a string.

the program in Listing 3. Initially, READ has a dependency on a_x. During the incremental update, the analysis of READ will find a new dependency on a_y, whilst the dependency on a_x can be removed.

Alg. 2 also extends INTER with DI. The set of dependencies computed during the last analysis of every component α, R_α, is cached using the cache $\mathbb{R}$ (also used by CI). After the (re-)analysis of a component α, INTER collects the computed dependencies, R'_α. It then fetches the dependencies computed during the *previous* analysis of α from $\mathbb{R}$ and computes the set of outdated dependencies which are then removed (lines 9-10). Finally, as for CI, $\mathbb{R}$ is updated (line 18).

3.4 Write Invalidation (WI)

Write invalidation (WI) aims to increase the precision of abstract values in the store. It is motivated by Listing 4. Variable `x` is changed from storing symbols to strings. A strong update would *overwrite* the abstract value `Symbol` by `String` in σ. A monotonic update instead joins the values together, resulting into the less precise value `{Symbol, String}`. Clearly, a strong update is desired.

The values in σ are part of an abstract domain, forming a complete lattice. The *higher* a value resides in the lattice, the less precise information it represents. WI aims to *lower* all values as much as possible by monitoring the values computed for every address in σ, and by lowering values that no longer correspond to the program's behaviour. We first describe the required monitoring.

Provenance Tracking Values in σ result from one or more writes, each monotonically updating the value. In this process, the analysis loses information w.r.t. the *constituents* and origins of the values. E.g., when α writes `1` to a and β writes

`-1` to a, $\sigma(a)$ contains `{Int}`, without information about the values written by α and β, nor about which components wrote these values. We introduce *provenance tracking* to regain this information. For every component and address in $dom(\sigma)$, the analysis maintains the *contribution* of the component to the address, i.e., the join of all values written to the address during the analysis of the component. This requires intercepting to *write* operations to the store.

Consider the case in Fig. 3: components α and β read and write two variables, `x` and `y`: both write `y`, α reads `x`, and β reads y. When α writes `Int` to `y` and β writes `Boolean` to `y`, σ holds join of these values, `{Int, Boolean}`, for `y`.

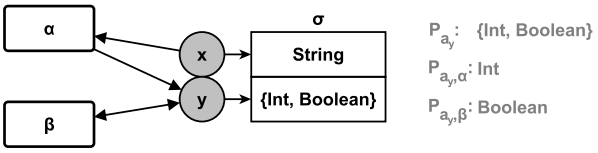

Fig. 3: Interaction of intra-component analyses with variables and their values in σ illustrated. On the right, the provenance and contributions of a_y are shown.

During the analysis of a component α, we track, for each written address a, the join of all values written to that address. We call this joined value the *contribution* of α to a, denoted $P_{a,\alpha}$. For every address, the contributions of all components are cached. We call this cache the *provenance* of the address, P_a. We define the *provenance value* of an address a as the join of all values in its provenance. Fig. 3 depicts this information on the right in grey.

Non-monotonic Store Updates The intra-component analyses perform all updates monotonically. Inter thus has to restore precision after it has been lost. Provenance tracking enables WI to perform non-monotonic updates to σ, improving its precision. This is possible when a previously-written address is no longer written by a component, and when the contribution of a component to an address changes in a non-monotonic way.[4] The code for WI is shown in Alg. 3.

Outdated writes. The analysis of a component tracks all addresses written to. For every component α, Inter caches this set, W_α, using a cache $\mathbb{W}$. After the analysis of a component α, Inter collects the set of written addresses, W'_α, and computes the set containing all addresses previously written by the component that are no longer written (line 26). Finally, the cache $\mathbb{W}$ is updated (line 28).

When the contribution $P_{a,\alpha}$ of α to an address a is removed, its provenance value, no longer influenced by $P_{a,\alpha}$, is used as the new value for the address (lines 2-3). If the provenance value equals the value at $\sigma(a)$, deletion is completed. Else, the provenance value replaces the value σ. All dependent components are

[4] Conceptually, the first case corresponds to the second case for which the contribution of the component to an address has become $\bot$. We treat it separately since no write to the address is performed any more.

scheduled for reanalysis (line 5), allowing the new value to be taken into account during their reanalysis, possibly leading to further refinements of the result. When an address is no longer written by any component, all information in the analysis' data structures related to this address can be removed (line 6).

More precise writes. After every intra-component analysis, INTER compares the contribution of the component for every written address, to the corresponding contribution computed by the component's previous analysis. Based on this comparison, the value at the given address in σ may be updated, in which case all dependent components are added to the worklist (line 29). The comparison may yield one of three possible results:

$P_{a,\alpha} = P'_{a,\alpha}$ The analysis did not compute new information, no information can be discarded (line 11).

$P_{a,\alpha} \sqsubset P'_{a,\alpha}$ The update is monotonic, no information can hence be discarded. The updated contribution is stored (line 12).

$P_{a,\alpha} \not\sqsubset P'_{a,\alpha}$ The contribution changes non-monotonically. The value for a can be replaced by the new provenance value (computed on line 14), now taking into account the updated contribution $P'_{a,\alpha}$ (stored in $\mathbb{P}$ on line 12).

The second and third case may not lead to an update of σ as the value computed on line 14 can be the same as the value already in σ. Only when the new value is different, dependent components need to be scheduled for reanalysis.

Reinforcing Component Invalidation §3.2 introduced CI. However, CI does not allow for the removal of information from σ: values written by removed components cannot be deleted, a limitation that can be remedied by combining CI with WI. When a component α is removed, all addresses in the set $\mathbb{W}(\alpha)$ are treated as outdated writes, described in §3.4. This allows σ to become more precise, which may in turn invoke the analysis of other components. The updated code for component deletion is shown in Algorithm 4.

4 Evaluation

We evaluated the presented strategies to answer the following research questions:

RQ1 How well do the three invalidation strategies improve the precision of the analysis, both when applied individually and when applied in combination?

RQ2 What is the impact of the invalidation strategies on the time needed to perform an incremental update?

RQ3 How much does the incremental analysis reduce the analysis time compared to a full reanalysis of the program?

We tested soundness of the initial analysis and the incremental update experimentally (1) by ensuring that the analysis over-approximates multiple runs of a concrete interpreter [1, 29], and (2) by comparing the incremental analysis results to the results of a non-incremental analysis. We performed these tests for a

Algorithm 3: INTER extended with write invalidation (in teal).

```
   // Assumes the existence of a cache for the sets W, 𝕎, initialised as 𝕎 := λα.∅
      before the initial analysis, and the existence of a cache ℙ, the provenance,
      initialised to ℙ := λa.(λα.⊥) before the initial analysis.
 1 Function deleteContribution(α, a) is
 2 |   ℙ := ℙ[a ↦ (ℙ(a) \ {α})];
 3 |   v := ⨆_{β∈dom(ℙ(a))} ℙ(a)(β);
 4 |   if v ≠ σ(a) then
 5 |   |   WL := WL ∪ D(a);
   |   |   // If an address is no longer written by any component, it is deleted.
   |   |      Otherwise, the store is updated.
 6 |   |   if ℙ(a) = ∅ then σ := σ \ {a}; ℙ := ℙ \ {a}; D := D \ {a}; else σ := σ[a ↦ v];
 7 |   end
 8 end
   // updateAddressIncremental compares the new contribution v′ of α to a to the
      previous contribution v, and improves the store if possible.
 9 Function updateAddressIncremental(α, a, v′) is
10 |   v := ℙ(a)(α); // Previous contribution of α to a, P_{a,α}.
11 |   if v = v′ then return false; // Identical contribution: no precision gain.
12 |   ℙ := ℙ[a ↦ (ℙ(a)[α ↦ v′])];
13 |   old := σ(a);
14 |   new := if v ⊑ v′ then old ⊔ v′ else ⨆_{β∈dom(ℙ(a))} ℙ(a)(β);
15 |   if old = new then return false;
16 |   σ := σ[a ↦ new]; // Update the store.
17 |   return true;
18 end
19 while WL ≠ ∅ do
20 |   ... // Ditto Alg. 1.
21 |   σ := σ′; // This line can now be omitted.
22 |   ... // Ditto Alg. 1.
23 |   foreach w ∈ W′ do WL := WL ∪ D(w); // This line can now be omitted.
24 |   if incremental update then
25 |   |   W := 𝕎(α);
26 |   |   foreach w ∈ (W \ W′) do deleteContribution(α, w);
27 |   end
28 |   𝕎 := 𝕎[α ↦ W′];
   |   // P computed during the intra-component analysis. P maps every written address
   |      to the join of all values written to it during the component's analysis.
29 |   foreach (a, v) ∈ P do if updateAddressIncremental(α, a, v) then WL := WL ∪ D(α);
30 end
31 return (σ, V, D);
```

thread-modular analysis for a concurrent Scheme, for a function-modular analysis, for all possible combinations of the invalidation strategies, and for a constant propagation and a type abstract domain; no unsound results were encountered.

4.1 Experimental Design

Our evaluation uses a context-insensitive ModF analysis for Scheme, with a LIFO-ordered worklist and a product lattice[5]. We implemented our contributions

[5] The lattice represents primitive values by their possible types, except booleans which are represented as their respective value when possible. Pointers are represented as sets of addresses (in $dom(\sigma)$); closures and primitives are represented using sets as well. A join of two values is the pointwise join of the corresponding elements of the product, where the join of two sets is their union.

Algorithm 4: `deleteComponent` (in blue) reinforced with WI (in teal).

```
1 Function deleteComponent(β) is
2     foreach r ∈ ℝ(β) do D := D[r ↦ D(r) \ {β}]; // Delete dependencies.
      // Remove β from all data structures.
3     V := V \ {β}; WL := WL \ {β}; ℝ := ℝ \ {β}; ℂ := ℂ \ {β};
4     W := 𝕎(β);
5     forall w ∈ W do deleteContribution(β, w);
6     𝕎 := 𝕎 \ {β};
7 end
```

in the open-source MAF framework[6] [29]. Our evaluation is run on a 2015 Dell PowerEdge R730[7] running OpenJDK 1.8.0_312 and Scala 3.1.0. The JVM was given a maximum of 32GB RAM, and all analyses used a timeout of 30 min.

To evaluate the precision of the incremental update (RQ1), we inspect the store σ at the end of the analysis. For each address, we measure the precision of the incremental update by comparing its value to its counterpart in the store of a full reanalysis. The proportion of addresses in the final store that contain values equally or less precise than the values obtained by a full reanalysis shows us how much precision can still be improved. We also compare to the store resulting from an incremental analysis without result invalidation. Here, the proportion of addresses in the final store that contain values equally or more precise than the values obtained by an incremental update without invalidation shows us how many addresses have an improved precision thanks to our strategies. We perform these comparisons for all possible combinations of the invalidation strategies.

To evaluate the performance of our strategies (RQ2 & RQ3), we measure the time needed to (1) analyse the initial program, (2) fully analyse the updated program, and (3) perform the incremental update given a set of enabled strategies. For (1) and (2), no strategy is enabled; the analysis will not maintain the caches required by any strategy. For (3), the initial analysis initialises all caches used by the strategies. Each measurement is repeated 15 times preceded by a warm-up of 3 repetitions or of maximally 30 min. Garbage collection is forced prior to each analysis.

Comparing the precision and performance of an incremental update using all strategies to (1) an (imprecise but fast) update without invalidation, (2) an update using only one or two strategies, and (3) a (precise but slow) full reanalysis, allows us to investigate a trade-off between precision and performance.

Benchmarking Suites Our evaluation uses two benchmarking suites.[8] Each benchmark program is a Scheme program containing real-world code, annotated with change expressions. As such, a benchmark corresponds to program changes.

[6] A repository containing our implementation can be found online: https://github.com/softwarelanguageslab/maf (branch `incremental-experiments`).

[7] The computer has 2 Intel Xeon 2637 processors and 256GB of RAM.

[8] In our online repository, the curated benchmarks can be found in the folders `/test/changes/scheme` and `/test/changes/scheme/reinforcingcycles`. The generated benchmarks can be found in the folder `/test/changes/scheme/generated`.

Curated Benchmarks. We curated a suite of 32 programs to which we manually added changes resembling possible developer edits, shown in Tab. 2. The programs originate from different sources, e.g., a university course with programming exercises in Scheme, together with the solutions for solving particular exercises, and benchmarking suites used by other researchers. Example edits include changing representations of data structures (e.g., replacing lists by vectors in `nbody-processed`), or updating a meta-interpreter (e.g., adding the ability to make variables immutable in `freeze` or making procedures dynamically scoped in `mceval-dynamic`). In programs like `slip-0-to-1`, `slip-1-to-2`, and `slip-2-to-3`, edits convert the program to a later version. A new abstraction is introduced and used throughout `peval`. Some edits were constructed to be tricky for an incremental update to process accurately, as they trigger *cyclic reinforcement of lattice values* [24, 23] (see §4.2). Also, certain programs contain the same changes but use a different granularity of change expressions; this is e.g., the case for `multiple-dwelling (coarse)` and `multiple-dwelling (fine)`, and for `satFine`, `satMiddle`, and `satCoarse`. The runtimes of the initial analyses of programs the curated suite vary from 0s to 117s.

Table 2: The curated suite, retrieved from various sources. For every benchmark, we list the lines of code as counted with `cloc` and the number of change expressions.

Benchmark	LOC	#Chg	Benchmark	LOC	#Chg
`baseline`	6	1	`primtest`	43	11
`browse`	164	1	`cycleCreation`	3	1
`collatz`	18	1	`higher-order-paths1`	4	2
`fact`	5	1	`higher-order-paths2`	4	1
`fib-loop`	15	1	`implicit-paths`	3	1
`fib`	5	2	`ring-rotate`	32	2
`freeze`	327	11	`sat`	16	4
`gcipd`	9	2	`satCoarse`	17	1
`leval`	379	11	`satFine`	13	3
`matrix`	617	3	`satMiddle`	16	3
`mceval-dynamic`	246	4	`satRem`	20	2
`multiple-dwelling (coarse)`	434	1	`slip-0-to-1`	123	6
`multiple-dwelling (fine)`	404	3	`slip-1-to-2`	117	3
`nbody-processed`	1252	10	`slip-2-to-3`	397	9
`nboyer`	636	2	`tab-inc`	317	3
`peval`	507	38	`tab`	307	3

Generated Benchmarks. We automatically generated 5 mutations for each of 190 programs, originating from various sources, obtaining 950 programs. We use a set of edit patterns of one or more change expressions that are inserted randomly, with a certain probability and at an arbitrary depth in the program. We consider the following patterns: expression deletion (7.5%), inserting a random

sub-expression (5%), swapping expressions (10%), wrapping an expression with a call to the identity function (7.5%), negating the predicate of an `if` (7.5%), and swapping the branches of an `if` (7.5%). A valid mutation has at least one edit, is unique, and does not lead to an error after running it with a Scheme interpreter for one minute. The runtimes of the initial analyses of programs the generated suite vary from 0s to 148s, most programs complete in under 10s.

4.2 Precision Evaluation (RQ1)

We evaluate the precision improvement caused by our invalidation strategies as follows. On every benchmark program, and for all possible configurations, we count the percentage of addresses in σ that is less precise than a full reanalysis. Fig. 4 depicts the results of our precision evaluation. These allow us to (1) evaluate the precision improvement caused by the application of the presented strategies, and (2) to see whether additional opportunities for precision improvement are possible. As a precision improvement of σ can only be expected when WI is enabled, we only show results for an incremental update without result invalidation, with WI, and with all strategies enabled (where CI reinforces WI).

Precision Improvements over Naive Incremental Analysis For the curated suite, in some cases such as `higher-order-paths1`, we observe a big precision improvement. On other programs, the improvement remains minor. `fib-loop` shows that reinforcing CI can lead to additional precision improvements. On benchmarks such as `browse` and `nbody-processed`, the benefit is smaller, though `browse` now reaches full precision. Unexpectedly, and only on `slip-0-to-1`, reinforcement decreases precision (this is not visible on the figure). The reason for this seems to be that, although sound, the obtained fixed-point depends on the analysis order of the components. On the generated suite, the number of imprecise values in the store is reduced by 15%-20% on average (geometric mean over all generated benchmarks): there is an improvement of about 10% with WI and an additional improvement of about 10% using all strategies.

Tab. 3 shows the quartiles of the distribution of the store's precision among all benchmarks in the generated suite for the same configurations. Without invalidation, more than 50% of all benchmark programs do not achieve full precision. However, using all strategies, the analysis reaches full precision on most benchmarks. The table shows the added benefit of reinforcing CI.

Remaining Imprecision in the Analysis Result Fig. 4 also shows remaining possibilities for precision improvement. On 13 curated benchmarks for which the incremental update without invalidation did not achieve full precision, the update with all strategies now does (indicated by a bar reaching 100%). However, on other benchmarks, more improvements remain possible.

The precision of σ influences the control flow explored by the analysis, and so the number of components and dependencies: precision gains due to WI can lead to the invalidation of components and dependencies when all strategies are

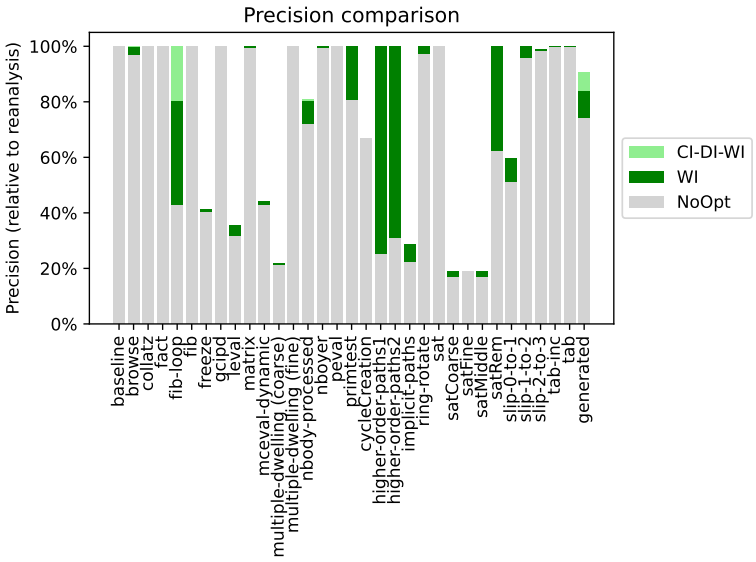

Fig. 4: Precision of values in σ after an incremental update compared to a full reanalysis. Bars represent the percentage of addresses in σ of an incremental update whose values match a full reanalysis. In grey, precision of an incremental update without invalidation is shown. In dark green, the additional percentage of matching addresses due to WI is shown. In light green, the further additional percentage of matching addresses using all strategies is shown. The rightmost bar shows the geometric mean of all benchmarks in the generated suite.

enabled. Of course, CI and DI can also be beneficial in without WI, though only WI can propagate precision gains to other components.

The imprecision in σ is worsened by our change representation: change expressions always require an old and new expression. For example, to introduce a new variable in a program, a placeholder value for the old program needs to be used, e.g., `#f` (false): `(define x (<change> #f 10))`. As this value will reside in σ and cannot be removed by the incremental update when WI is not enabled, some values in σ may be artificially imprecise. However, imprecision still remains for some benchmarks when WI is enabled. One reason we found is *cyclic reinforcement of lattice values* [24, 23], which arises when, due to the abstractions in the analysis, the computation of a value at an address is influenced by the value at that address itself, thereby influencing its own provenance.[9] WI cannot restore the precision of values in such a cycle. We also believe that this phenomenon causes the result to depend on the exploration order, e.g., when

[9] Some programs in our curated suite, such as `cycleCreation` and `implicit-paths`, are explicitly created to contain this behaviour.

Table 3: Precision of values in σ after an incremental update compared a full reanalysis. Percentages indicate the number of addresses in σ of an incremental update whose values match a full reanalysis. The table shows the quartiles of the distribution of these percentages among all programs in the generated suite, for an incremental analysis without invalidation, with WI, and with all strategies.

Configuration	**Q1**	**Q2**	**Q3**
No invalidation	73%	98%	100%
WI	97%	100%	100%
CI-DI-WI	100%	100%	100%

a value is refined before being introduced into a cycle, the cycle will be more precise than when refining would have taken place afterwards.

Answer RQ1. Only WI can improve the precision of σ. WI significantly improves the precision of values for a limited number of curated benchmarks. Maximal precision is reached for 13 extra benchmarks when using all strategies, i.e., using reinforced CI. For other curated benchmarks, a large percentage of addresses remains less precise. We also observe a big improvement on the generated suite, though several addresses still remain imprecise. Once again, the combination of CI and WI leads to a substantial additional precision improvement.

4.3 Performance w.r.t. No Invalidation (RQ2)

Fig. 5 shows the results of the performance evaluation for RQ2. Times are shown relative to an incremental update without invalidation. CI and DI do not cause a significant slowdown of the incremental analysis. A slowdown appears when using WI, but, overall, the incremental update remains faster than a full reanalysis (see §4.4). This slowdown can be explained as follows. As WI refines σ, updates may trigger the reanalysis components, leading to further reanalyses and impacting performance. On the curated benchmarks, this increase in running time is more moderate for the combination of CI and WI.

CI and WI combined reduce, in some cases, the analysis time as outdated components are not analysed anymore. Also, WI may create more opportunities for CI: when values become more refined, this may lead to more outdated components, which may in turn lead to an improvement of values in σ.

Answer RQ2. CI and DI have no substantial negative impact on the running time of an incremental update. Only WI causes a slowdown: as WI regains precision, changes to σ may cause components to be scheduled for reanalysis.

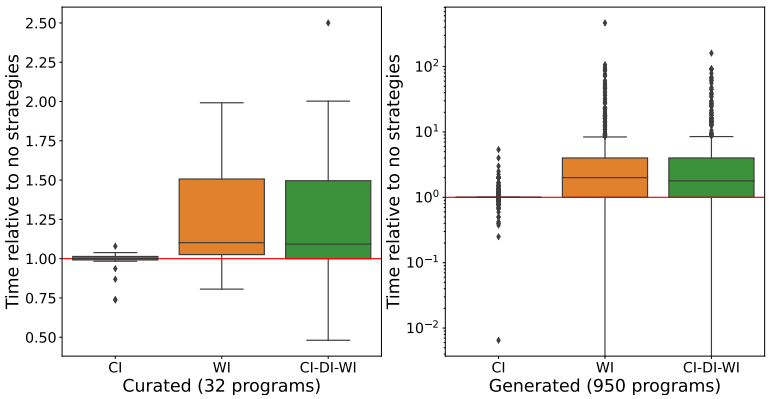

Fig. 5: Analysis time of the incremental update relative to an incremental update without invalidation. Benchmarks for which the incremental update completed in 0ms are omitted in the graphs, because a relative time cannot be computed.

4.4 Performance w.r.t. Full Reanalysis (RQ3)

Fig. 6 shows the results of our performance evaluation for RQ3. Times are shown relative to the time needed by a full reanalysis.

For the curated suite, overall, the incremental update is faster than a full reanalysis. The medians are consistently under 0.2, meaning that on more than half of the benchmark programs, the incremental update is more than 5 times faster. When both CI and WI are used, we see one outlier which corresponds to the `primtest` benchmark for which the running times are very low, meaning that there is no opportunity for the incremental analysis to gain time.

The results of the generated suite are grouped based on the time taken by the initial analysis and the full reanalysis. The slowdown caused by WI is most outspoken for short-running generated benchmark programs, where the overhead of the strategies may be relatively high. When both the initial analysis and full reanalysis complete in under a second, and when both analyses run a second or longer, overall, the incremental update remains faster than a full reanalysis. Although WI may cause minor slowdowns, the incremental update remains more than 10× faster compared to a full reanalysis. On programs that have an initial analysis taking a second or more but a shorter full reanalysis, the incremental update is slower: for almost all configurations, the incremental analysis takes at least as long as a full reanalysis for most benchmarks, with median slowdowns of up to 100 and outliers showing slowdowns larger than 1000. It is difficult to pinpoint the exact root cause for each performance difference. We list several possible reasons that may explain this behaviour:

– The change representation may cause less result reuse. In our implementation, change expressions cannot be placed at all program points. Some changes must be represented with coarse-grained change expressions. E.g., to rename a

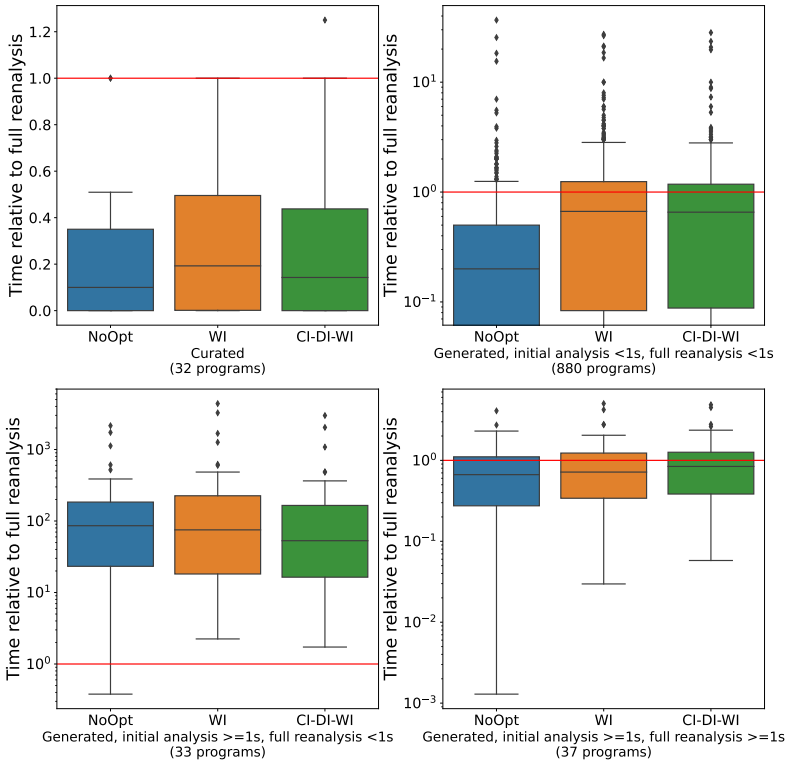

Fig. 6: Analysis time of the incremental update relative to a full reanalysis. Benchmarks for which the full reanalysis completed in 0ms are counted but omitted in the graphs because a relative time cannot be computed.

function parameter, the change expression must wrap around the entire function definition, thus components corresponding to the function cannot be reused.

– The generated programs may contain too many changes, leading to many impacted components: 25 programs have over 30 changes and 79 programs have over 20 changes. On almost half of the programs, more than 20% of the components is directly affected. As many components are affected, the incremental analysis may not benefit from its modularity to bound the impact of the changes.

– Changes may significantly alter program behaviour. 33 benchmarks had a long-running initial analysis and short-running full reanalysis. In these cases, the incremental analysis performs very poorly. It is possible that the randomly inserted changes prune away a lot of program functionality, leading to a very fast reanalysis, whereas an incremental update needs to propagate information deletion. Although we haven't verified the behaviour of all benchmarks individually, the reduced running time of the full analysis indicates that in these cases, an incremental update is inadequate due to the nature of the program changes.

– No dedicated worklist algorithm is used. Components may be scheduled for analysis due to newly inferred information or due to invalidation, but neither is prioritised. By intertwining recomputation by invalidation, information may be added or removed in an unspecified order; information may be removed that is later readded, or vice versa. We assume that the analysis of components in an unordered way may negatively impact the analysis performance.

To improve performance, future work should consider imposing an order on the worklist. It may also be useful to investigate heuristics to determine which changes would better be processed by a full reanalysis, e.g., when a program update leads to a big removal of program functionality.

Answer RQ3. On the curated suite, on the short-running generated benchmarks, and on the generated benchmarks with a long-running initial analysis and reanalysis, overall, the incremental update is faster. Yet, on the generated benchmarks with a long-running initial analysis but with a short-running full reanalysis, almost all incremental updates are slower. The nature of the changes may be to blame for this: a very fast reanalysis may indicate a serious reduction in program behaviour, in which case the incremental update has to invalidate many results, causing high relative runtimes.

5 Related Work

Nichols et al. [13] introduce *fixpoint reuse* to incrementally analyse JavaScript programs. They map program points to corresponding program points in the new program, allowing reuse of analysis results for mapped points. The mapping function plays a key role: more mapped points lead to more reuse and a faster analysis, but incorrect matches can cause the analysis to lose precision.

IncA [27, 24, 26, 25] is a Datalog-based analysis framework that produces the same results as a full reanalysis. It uses an incremental Datalog with a semi-naïve, stratified evaluation strategy [25]. For every tuple, a *support count* indicating the number different derivations of the tuple is maintained and used to invalidate tuples after program updates. Contrary to IncA, our approach does not require programs and analyses to be converted into a Datalog-like representation.

Andromeda [28] is an incremental, demand-driven taint analysis. Its relies on a *support graph* to find taint facts that are outdated. In contrast, our analysis is not tailored to a specific client analysis. Saha and Ramakrishnan [20] also use support graphs in their framework for implementing incremental, demand-driven analyses. They require analyses and programs under analysis to be specified as Horn clauses and represent changes by means of the addition or deletion of facts.

Reviser [2] is an incremental, inter-procedural data-flow analysis for analyses expressible in the IDE or IFDS frameworks. Its results match a full program analysis but it requires a static call-graph; dynamic languages are unsupported. Our approach is not limited to specific analyses and does not require a static call graph. Other incremental approaches relying on static call graphs comprise a.o. alias analyses [33], interval analyses [3], dataflow analyses [34, 4, 19], and analyses

tailored to specific client tools, such as race detection [35]. Liu et al. [11] present an incremental points-to analysis not requiring a prebuilt call graph. It preserves precision but is limited to flow-insensitive analyses, unlike ours.

Garcia-Contreras et al. [8] present a context-sensitive incremental modular analysis which achieves incrementality at the inter-modular and intra-modular level. The analysis requires an encoding of the program in constrained Horn clauses. Contrary to ours, the analysis does not divide the program into modules itself and does not use components but a programmer-defined lexical module partitioning is used, but it is claimed that any partitioning is possible. Thus, their analysis can, e.g., not be used with thread-modular analyses, in contrast to ours. Later work [9] presents an updated approach, also capable of handling external modules, together with a formal description and a further evaluation.

6 Conclusion

We presented three complementary invalidation strategies to improve the precision and performance of the incremental modular analysis approach presented by Van der Plas et al. [18]. Our approach interleaves reanalysis of components with invalidation. Component invalidation removes outdated components and their dependencies, and, when combined with write invalidation, can also improve the precision of the values in the store σ. Dependency invalidation removes outdated dependencies. Write invalidation uses provenance tracking to retract and replace outdated contributions from components to σ, enabling non-monotonic updates.

We tested our strategies for unsoundness and evaluated their precision and performance empirically on real-world programs using a small suite of 32 programs with possible developer edits and a large corpus 950 of programs with generated edits. Our strategies allow the incremental analysis to reach the same result as a full reanalysis on 13 more programs in the curated suite in comparison to when none of the proposed strategies is used. On other programs, the precision loss is reduced, yet the results did not match the precision of a full reanalysis. For the generated suite, using all strategies, on average, the number of less precise addresses in σ is reduced from 30% to about 10%. The best improvements were realised by the combination of write invalidation with component invalidation.

Performance-wise, overall, the incremental analysis scores well. We did find some benchmarks with particular program changes for which the incremental update proved to be slower than a full reanalysis, e.g., in 33 of the 950 programs in the generated suite where the changes removed a big part of a program's functionality. Future work includes handling cyclic reinforcement of lattice values, stratifying the worklist of the analyses, and investigating heuristics for triggering a full reanalysis rather than an incremental update.

Acknowledgements

This work was partially supported by the Research Foundation – Flanders (FWO) (grant number 11F4822N) and by the *Cybersecurity Initiative Flanders*.

References

1. Andreasen, E.S., Møller, A., Nielsen, B.B.: Systematic Approaches for Increasing Soundness and Precision of Static Analyzer. In: Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP 2017. pp. 31—36. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3088515.3088521
2. Arzt, S., Bodden, E.: Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, May 31-June 07, 2014. pp. 288–298. ACM Press, New York, NY, USA (2014). https://doi.org/10.1145/2568225.2568243
3. Burke, M.G.: An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. ACM Trans. Program. Lang. Syst. **12**(3), 341–395 (1990). https://doi.org/10.1145/78969.78963, https://doi.org/10.1145/78969.78963
4. Carroll, M.D., Ryder, B.G.: Incremental data flow analysis via dominator and attribute updates. In: Ferrante, J., Mager, P. (eds.) Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988. pp. 274–284. ACM Press (1988). https://doi.org/10.1145/73560.73584, https://doi.org/10.1145/73560.73584
5. Cousot, P., Cousot, R.: Modular Static Program Analysis. In: Horspool, R.N. (ed.) Proceedings of the 11th International Conference on Compiler Construction, CC 2002, Grenoble, France, April 8-12, 2002. pp. 159–178. Springer, Berlin, Heidelberg, Germany (2002). https://doi.org/10.1007/3-540-45937-5_13
6. Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and Accurate Source Code Differencing. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15-19, 2014. pp. 313–324. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642982
7. Gall, H.C., Fluri, B., Pinzger, M.: Change Analysis with Evolizer and ChangeDistiller. IEEE Softw. **26**(1), 26–33 (2009). https://doi.org/10.1109/MS.2009.6
8. Garcia-Contreras, I., Caballero, J.F.M., Hermenegildo, M.V.: An Approach to Incremental and Modular Context-Sensitive Analysis (2018), http://oa.upm.es/53067/
9. Garcia-Contreras, I., Morales, J.F., Hermenegildo, M.V.: Incremental and Modular Context-sensitive Analysis. Theory and Practice of Logic Programming **21**(2), 196–243 (2021). https://doi.org/10.1017/S1471068420000496
10. Hattori, L., Lanza, M.: Syde: A tool for collaborative software development. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2. p. 235–238. ICSE 2010, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1810295.1810339
11. Liu, B., Huang, J., Rauchwerger, L.: Rethinking Incremental and Parallel Pointer Analysis. ACM Transactions on Programming Languages and Systems **41**(1), 6:1–6:31 (2019)
12. Negara, S., Vakilian, M., Chen, N., Johnson, R.E., Dig, D.: Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In: Noble, J. (ed.) Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP 2010, Beijing, China, June 11-16, 2012. pp. 79–103. Springer (2012). https://doi.org/10.1007/978-3-642-31057-7_5

13. Nichols, L., Emre, M., Hardekopf, B.: Fixpoint reuse for incremental javascript analysis. In: Grech, N., Lavoie, T. (eds.) Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019. pp. 2–7. ACM (2019). https://doi.org/10.1145/3315568.3329964, https://doi.org/10.1145/3315568.3329964
14. Nicolay, J., Noguera, C., De Roover, C., De Meuter, W.: Determining dynamic coupling in javascript using object type inference. In: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 126–135. IEEE (2013)
15. Nicolay, J., Stiévenart, Q., De Meuter, W., De Roover, C.: Purity analysis for JavaScript through abstract interpretation. Journal of Software: Evolution and Process **29**(12), e1889 (2017)
16. Nicolay, J., Stiévenart, Q., De Meuter, W., De Roover, C.: Effect-Driven Flow Analysis. In: Enea, C., Piskac, R. (eds.) Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2019, Cascais, Portugal, January 13-15, 2019. pp. 247–274. Springer International Publishing, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-030-11245-5_12
17. Palikareva, H., Kuchta, T., Cadar, C.: Shadow of a Doubt: Testing for Divergences Between Software Versions. In: Dillon, L.K., Visser, W., Williams, L. (eds.) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. pp. 1181–1192. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884845
18. Van der Plas, J., Stiévenart, Q., Van Es, N., De Roover, C.: Incremental Flow Analysis through Computational Dependency Reification. In: 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, September 27-28, 2020. pp. 25–36. IEEE Computer Society (2020). https://doi.org/10.1109/SCAM51674.2020.00008
19. Pollock, L.L., Soffa, M.L.: An incremental version of iterative data flow analysis. IEEE Trans. Software Eng. **15**(12), 1537–1549 (1989). https://doi.org/10.1109/32.58766, https://doi.org/10.1109/32.58766
20. Saha, D., Ramakrishnan, C.R.: Incremental and Demand-driven Points-To Analysis Using Logic Programming. In: Barahona, P., Felty, A.P. (eds.) Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal. pp. 117–128. ACM (2005). https://doi.org/10.1145/1069774.1069785
21. Stievenart, Q., Nicolay, J., De Meuter, W., De Roover, C.: Detecting concurrency bugs in higher-order programs through abstract interpretation. In: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming. pp. 232–243 (2015)
22. Stiévenart, Q., Nicolay, J., De Meuter, W., De Roover, C.: A General Method for Rendering Static Analyses for Diverse Concurrency Models Modular. Journal of Systems and Software **147**, 17–45 (2019). https://doi.org/10.1016/j.jss.2018.10.001
23. Szabó, T.: Incrementalizing Static Analyses in Datalog. Doctoral dissertation, Johannes Gutenberg-Universität Mainz, Mainz, Germany (2021). https://doi.org/http://doi.org/10.25358/openscience-5613
24. Szabó, T., Bergmann, G., Erdweg, S., Voelter, M.: Incrementalizing lattice-based program analyses in Datalog. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–29 (2018). https://doi.org/10.1145/3276509

25. Szabó, T., Erdweg, S., Bergmann, G.: Incremental Whole-Program Analysis in Datalog with Lattices. In: Freund, S.N., Yahav, E. (eds.) Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021. pp. 1–15. ACM, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454026
26. Szabó, T., Bergmann, G., Erdweg, S.: Incrementalizing inter-procedural program analyses with recursive aggregation in Datalog p. 3 (2019), Presented at the Second Workshop on Incremental Computing, IC 2019, Athens, Greece, October 21, 2019
27. Szabó, T., Erdweg, S., Voelter, M.: IncA: A DSL for the Definition of Incremental Program Analyses. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016. pp. 320–331. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2970276.2970298
28. Tripp, O., Pistoia, M., Cousot, P., Cousot, R., Guarnieri, S.: Andromeda: Accurate and Scalable Security Analysis of Web Applications. In: Cortellessa, V., Varró, D. (eds.) Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013, Rome, Italy, March 16-24, 2013. pp. 210–225. Springer, Berlin, Heidelberg, Germany (2013). https://doi.org/10.1007/978-3-642-37057-1_15
29. Van Es, N., Van der Plas, J., Stiévenart, Q., De Roover, C.: MAF: A Framework for Modular Static Analysis of Higher-Order Languages. In: 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020. IEEE Computer Society (2020)
30. Van Horn, D., Might, M.: Abstracting Abstract Machines. In: Hudak, P., Weirich, S. (eds.) Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, MD, USA, September 27-29, 2010. pp. 51–62. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863543.1863553
31. Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H.C., Zaidman, A.: How developers engage with static analysis tools in different contexts. Empirical Software Engineering **25**(2), 1419–1457 (2020). https://doi.org/10.1007/s10664-019-09750-5
32. Yoon, Y., Myers, B.A.: Capturing and analyzing low-level events from the code editor. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools. p. 25–30. PLATEAU '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2089155.2089163, https://doi.org/10.1145/2089155.2089163
33. Yur, J., Ryder, B.G., Landi, W.: An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) Proceedings of the 1999 International Conference on Software Engineering, ICSE 1999, Los Angeles, CA, USA, May 16-22, 1999. pp. 442–451. ACM (1999). https://doi.org/10.1145/302405.302676
34. Zadeck, F.K.: Incremental Data Flow Analysis in a Structured Program Editor. In: Deusen, M.S.V., Graham, S.L. (eds.) Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984. pp. 132–143. ACM (1984). https://doi.org/10.1145/502874.502888
35. Zhan, S., Huang, J.: ECHO: Instantaneous In Situ Race Detection in the IDE. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, Novem-

ber 13-18, 2016. pp. 775–786 (2016). https://doi.org/10.1145/2950290.2950332, https://doi.org/10.1145/2950290.2950332